Model Checking Semi-Continuous Time Models Using BDDs

Sérgio Campos, [a] Márcio Teixeira, [a] Marius Minea, [b]
Andreas Kuehlmann, [c] Edmund Clarke [b]

[a] *Univ. Federal de Minas Gerais, Dept. de Ciência da Computação, Brasil*
{scampos,mto}@dcc.ufmg.br

[b] *Carnegie Mellon University, School of Computer Science, USA*
{marius,emc}@cs.cmu.edu

[c] *IBM T.J. Watson Research Center, USA*
kuehl@watson.ibm.com

Abstract

The verification of timed systems is extremely important, but also extremely difficult. Several methods have been proposed to assist in this task, including extensions to symbolic model checking. One possible use of model checking to analyze timed systems is by modeling passage of time as the number of taken transitions and applying quantitative algorithms to determine the timing parameters of the system. The advantage of this method is its simplicity and efficiency. In this paper we extend this technique in two ways. First, we present new quantitative algorithms that are more efficient than their predecessors. The new algorithms determine the number of occurrences of events in all paths between a set of starting states and a set of final states. We then use these algorithms to introduce a new model of time, in which the passage of time is dissociated from the occurrence of events. With this new model it is possible to verify systems that were previously thought to require dense time models. We use the new method to verify two such examples previously analyzed by the HyTech tool: a steam boiler example and a fuel injection controller.

1 Introduction

Computers are frequently used in applications where failures can have severe consequences, such as in the control of industrial machinery or transportation equipment. In these applications, the computer system must not only produce the correct result, but must do so in timely fashion. For example, a command to apply the brakes of a car or to turn an airplane to a certain direction cannot be late, otherwise an accident may occur. Such failures cannot be tolerated, making the correctness of these systems an extremely important issue.

However, verification of such systems is a very complex problem, made even harder by timing requirements. Several methods have been proposed to accomplish this task. One method that has obtained significant success is model checking [7,8]. In this technique the system being verified is modeled as a state-transition graph and properties of the system are expressed as temporal logic formulas. The verification procedure consists of a search on the state space of the graph to determine which states satisfy the properties.

Original model checkers were not designed to verify timing characteristics. Several extensions have been proposed to express and verify such properties. The first and simplest is to associate each transition with the passage of one time unit and to determine elapsed time by counting the number of transitions between events. This technique assumes a *discrete time* model. The main advantage is its simplicity and extremely efficiently implementation, particularly in BDD-based symbolic model checkers such as SMV [16] or Verus [4].

Another approach is to use a *continuous time* model, in which events can happen at any moment in the dense time domain, e.g., timed automata [1,10]. Since in this case the state space is inherently infinite, model checking entails constructing a finite equivalent model, the complexity of which can be quite high. These models, as well as the verification algorithms are considerably more complex than in the discrete time case. Initial tools were unable to handle models with more than hundreds or thousands of states. Current tools are significantly more efficient [11,15,18], but verifying timed automata is still much more expensive than the verification of discrete time models.

However, discrete time models have one major disadvantage over continuous time models: their limitation in expressing the semantics of event sequences that happen in short periods of time. For example if the occurrence of an event a triggers an alarm b and an immediate response c we can model these events as happening simultaneously or taking at least two time units to occur. This may not correspond to reality, however. It may be the case that after event a has occurred but before alarm b another event d occurs that would change response c. But if a, b and c happen at the same time this possibility would not be present. On the other hand, if it takes 1 time unit between a and b it would not be possible for d to occur *between* a and b. For this reason discrete time models cannot be used in some applications where accuracy is essential.

The proposed method overcomes this problem by using *zero-length* transitions to model the occurrences of events without time passing. The passage of time then occurs in discrete steps using unit-length transitions. The advantage of this new model is that it removes the limitation on event orderings for the discrete time model. For example, it is now possible to let events a, b, c and d described previously occur in time zero preserving their order, and only let time elapse after all events have occurred. We argue that this enables the verification of many systems that have been previously thought to require dense time models.

In order to determine the time between events in the semi-continuous time model we use quantitative timing analysis as described in [6,3,4]. Of particular interest are the *condition counting* algorithms that count the minimum and maximum number of occurrences of a specific event in a given set of intervals. In this work these algorithms are used to count the minimum and maximum number of unit transitions on paths of interest, computing the time elapsed between events. We propose new condition counting algorithms that are significantly more efficient than the previous ones. These algorithms allow verification to be done as efficiently as for the simple discrete time case. They are similar to the fixpoint computation used in model checking for untimed systems, and as such can be implemented efficiently using BDDs.

To demonstrate the expressive power and efficiency of the method we have verified two examples of systems in which high accuracy is necessary to achieve the correct results. The first is the steam boiler example described in [14]. This example, while small, demonstrates that the proposed model can be used to verify systems which are not usually considered in the realms of discrete time. We have then verified an automotive engine controller developed for Magneti-Marelli that has been previously verified by HyTech [17]. We have modeled the controller that identifies that the driver has released the accelerator and regulates the reduction of fuel injection. This identification is a complex time critical function of the position of several sensors. If the timing of the events that take place during its execution is wrong, the algorithm may not converge and the controller can malfunction. We have verified both examples using Verus, demonstrating the effectiveness of the proposed method.

2 Related Work

A precursor to the presented analysis method has been developed in the real-time model checker Verus [4]. This tool implements *quantitative timing analysis* algorithms that determine the timing characteristics of a system by counting the time between events or the number of occurrences of events in given intervals. The method has been used to verify large and complex timed systems such as an aircraft controller [6], a robotics controller [5] and the PCI local bus [3]. However, the condition counting algorithms used in that context require the augmentation of the state space with a additional integer time variable which added a significant overhead to verification. The new algorithms do not require this construct and are efficiently implemented using BDDs.

The occurrence of events without the passage of time has been discussed in [9]. But that work does not consider a symbolic implementation using BDDs and is not as efficient. It also does not use quantitative analysis algorithms and cannot generate the same type of information as the method proposed.

A significant body of research exists on continuous-time models. One of the most widely used models are the *timed automata* [1], which add real-valued *clock variables* to represent time. Clocks evolve at the same rate, modeling

passage of time, and formulas can refer to the value of the clocks to express timing properties. Verification is then performed on a finite-state quotient model, such as the region graph [1] or the zone automaton [13]. However, the expressive accuracy comes with a significant increase in complexity, and a significant effort in the development of continuous-time model checkers [18,15] has been devoted to dealing with the state explosion problem.

The expressiveness and efficiency trade-offs between discrete and continuous time raise the question when a discrete-time approximation is sufficient to model all continuous-time behaviors of a system. This problem is analyzed, e.g., in [12]. This work introduces the notion of *digitizability* and proves that such a reduction is possible for timed transition systems, for verification of properties such as time-bounded invariance and time-bounded response. More recent work [2] shows that a reduction to discrete time can be performed for acyclic combinational circuits, but not for all cyclic ones. These can only be reduced under the constraint that no strict inequality is used in their design.

3 Condition Counting Algorithms

Our method relies on the ability to count *some* transitions on a path but not necessarily all of them. In order to accomplish this, we use the algorithms described in this section. The original algorithms used in our method to verify real-time systems determine the length of a path leading from a set of starting states to a set of final states [6,3,5]. But to verify semi-continuous time models we also need to compute the minimum and maximum number of times a given condition holds on any path from *start* to *final*. In [6] we have presented algorithms that compute this information. However, these algorithms required an augmentation of the state space with a counter to store intermediate results. This made the algorithms very expensive in some cases. The algorithms described in this section do not suffer from this limitation.

We require that every state of the model has at least one outgoing transition. We also assume that any path beginning in *start* reaches a state in *final* in a finite number of steps. This is necessary so that the minimum and maximum are well-defined. It can be checked using the maximum algorithm described in [6]. We also consider only reachable states, which can be achieved by intersecting *start* with the set of reachable states computed a priori.

Minimum Condition Counting

The minimum condition count algorithm computes the minimum number of states satisfying a given condition *cond* over all paths that start in a state in *start* and end in a state in *final*. Any paths starting in *start*, but which do not reach *final* in a finite number of steps are excluded from this computation. In particular, if no path from *start* ever reaches *final*, the algorithm will return the special value `NOPATH`.

The algorithm looks for paths beginning in *start* that have an increasing number of occurrences of *cond*. Each iteration consists of two phases: The first is a forward traversal through states that do not satisfy *cond*. This traversal is performed until all states (not satisfying *cond*) reachable from the current frontier are found. If *final* has not been reached yet, the frontier is expanded by one step to states that satisfy *cond* and the condition counter is incremented. The algorithm iterates until *final* is found, or all reachable states are visited.

The algorithm must differentiate between states that do not satisfy *cond* and those that do, and similarly, between transitions leading to these states. We use subscripts 0 and 1 respectively for the two types of states and transitions. For example, $start_0$ is the set of initial states that do not satisfy *cond*, and $start_1$ is the set of initial states that satisfy *cond*:

$$start_0 = start \cap \neg cond \qquad start_1 = start \cap cond$$

Furthermore, if $N(s, s')$ is the transition relation, we denote by $T_0(S)$ and $T_1(S)$ the set of transitions from a state in S that lead to states not satisfying *cond* and to states satisfying *cond*, respectively:

$$T_0(S) = \{s' \mid \exists s \in S.N(s, s') \wedge s' \notin cond\}$$
$$T_1(S) = \{s' \mid \exists s \in S.N(s, s') \wedge s' \in cond\}$$

The argument about the correctness of the algorithm follows from invariants stating that R' at the i^{th} iteration contains the set of all states that can be reached as endpoints of finite intervals starting in *start*, have no state in *final* (except perhaps the last one), and having i or less states satisfying condition. The proof can be found in the full version of the paper.

Maximum Condition Counting

The maximum condition count algorithm computes the maximum number of states satisfying a given condition *cond* over all paths that begin in a state in *start* and end in a state in *final* without previously traversing a state in *final*. If there is a path beginning in *start* that goes through *cond* infinitely often without reaching *final*, the algorithm returns infinity. The basic idea behind the algorithm is to find paths with increasing condition count whose states are all within $\neg$*final*. The condition count of the longest path satisfying this condition and starting in *start* is the desired maximum.

Similarly to the mincount algorithm, we consider transitions into states that satisfy *cond* and that do not satisfy *cond* separately. This algorithm, however, performs a backward search, and uses the reverse image of the transition relation. In this case $B_0(S')$ is the set of states satisfying neither *cond* nor *final* that lead to a state in S' in one step. Similarly, $B_1(S')$ is the set of states satisfying *cond* but not $final$ that lead to a state in S' in one step. Note that *final* only appears implicitly in the algorithm, in the definitions of B_0 and B_1.

$$B_0(S') = \{s \mid \exists s' \in S'.N(s, s') \wedge s \notin final \wedge s \notin cond\}$$
$$B_1(S') = \{s \mid \exists s' \in S'.N(s, s') \wedge s \notin final \wedge s \in cond\}$$

```
proc mincount(start, cond, final)
i = 0; R = ∅; R' = start_0;
do
  do
    if (R' ∩ final ≠ ∅) return i;
    R = R';
    R' = T_0(R') ∪ R';
  while (R' ≠ R);
  R' = T_1(R') ∪ R';
  if (i = 0)R' = R' ∪ start_1;
  i = i + 1;
while (R' ≠ R);
return NOPATH;

proc maxcount(start, cond, final)
i = 0; R' = cond;
do
  R_1 = R';
  do
    R = R';
    R' = R' ∪ B_0(R');
  while (R' ≠ R);
  if (R' ∩ start = ∅) return i;
  R' = B_1(R');
  i = i + 1;
while (R' ≠ R_1);
return ∞;
```

Fig. 1. Minimum and maximum condition count algorithms

Again, we argue the correctness of the algorithm using an invariant similar to the previous one. It states that at the i^{th} iteration R' is the set of all states that are the start of a finite path which has no states in *final* (except possibly the last one), and which has $i+1$ states that belong to *cond*. The proof can be found in the full version of the paper.

4 Semi-Continuous Time

The basic idea of the proposed method is to allow zero-length transitions that model the occurrence of events without time passing, thus making the occurrence of events independent of the passage of time. To allow zero-length transitions we have created a special variable t in the model of the system being verified. Time passage is controlled by enabling unit-length transitions only when t is $true$, and enabling zero-length transitions only when t is $false$.

Parallel composition of processes under the new model is defined as follows. Unit transitions have to occur synchronously, that is, all processes must

execute a unit transition in order for time to elapse. Zero-length transitions, on the other hand occur asynchronously. When a process performs a zero-length transition all other processes are not executing. As a consequence of this, zero-length transitions are always enabled. Unit transitions however, are only enabled when there is at least one unit transition enabled in each process. This parallel composition model satisfies one important invariant: passage of time is identical in all processes.

A symbolic implementation of this parallel composition model is straightforward given the traditional parallel composition algorithms used in BDD-based tools: conjunction of transition relations for synchronous composition and disjunction for asynchronous composition.

Under the new model we must first differentiate between unit and zero-length transitions. Given TR_a we define $TR0_a$ ($TR1_a$) as the transition relation for zero-length (unit) transitions in P_a. We can then define the global transition relation for a model with processes P_a and P_b as:

$$TR = (TR1_a \wedge TR1_b) \vee (TR0_a \vee TR0_b)$$

From this expression we can see that whenever unit transitions are enabled in all processes they are also enabled in the composed model. The expression also guarantees that zero-length transitions enabled in some process are also always enabled in the composed model. The only other condition that must be imposed in this model is that time eventually change. This can be ensured by forbidding zero-length loops, which can be enforced by a syntactic check.

To determine how much time has elapsed between events, we use the condition counting algorithms. For example, `mincount`$[a, t = true, b]$ determines the minimum time between events a and b. Similarly the `maxcount` algorithm can be used to determine the longest time between a and b.

5 Expressive Power of the Proposed Method

The proposed method does not have the same expressive power as a dense time model. Our method uses a different "discretization" of dense time, but the final model is still discrete. It has been proven [2] that there exist systems which cannot be discretized without changing their behavior. In [2] it is shown that the following circuit has behaviors that cannot be captured by any discretization. It has four signals x_0, x_1, x_2 and x_3, and transitions which assign values to them as: $x_1 = \neg x_0, x_2 = \neg x_0$ and $x_3 = \neg x_0$. Each transition takes time between 0 and 1 units to occur. Let t_1, t_2 and t_3 be the times when each transition occurs. A possible behavior of the circuit could have transitions times satisfying $0 \leq t_1 < t_2 < t_3 \leq 1$. In a discrete time model the only values allowed for t_i are 0 or 1, it is impossible to assign three different values for t_1, t_2 and t_3. This behavior cannot exist in a discrete model.

The result of [2] is that only models without strict inequalities can be guaranteed to be discretized correctly. Only a weaker notion of behavior preservation can be maintained during discretization: It is possible that events

that occur at different time instants in the dense time model occur at the same time instant in the discretized model. This is also true for our model. It is frequently argued that because of this problem systems modeled using discrete time cannot capture the essential properties of a design. We argue, however, that the key feature is not an arbitrary accuracy for the representation of t_1, t_2 and t_3, but rather an appropriate discretization together with their ordering. In fact, in the commonly used continuous-time models, the constants used in specifying properties can only be integers, and exact values for the timepoints t_i are not expressible.

With the use of transitions that take zero time to occur, our method provides a way of preserving the same ordering of events as dense time models. We claim then that the essential properties of a design are preserved by our method in a similar way as by methods that use dense time. For example, one property that would capture the behavior above can be written in CTL augmented with the *freeze operator* "." described in [9] as (where e_i is the event corresponding to the transition of signal x_i):

$$x.(e_1 \rightarrow EF\ y.(e_2 \rightarrow EF\ z.(e_3 \wedge 0 \leq x < y < z \leq 1)))$$

This can be expressed in our method by the property:

$$(e_1 \wedge \neg e_2) \rightarrow E[\neg t\ U\ (e_2 \wedge \neg e_3 \wedge \neg t \wedge E[\neg t\ U\ e_3])]$$

where t is true in unit-length transitions, and false in zero-length ones.

Frequently, the fact that the total time elapsed is less than one time unit is not encoded in the formula. In this case the formula can be simplified to

$$(e_1 \wedge \neg e_2) \rightarrow EF(e_2 \wedge \neg e_3 \wedge EFe_3)$$

One important consideration is that this property can be verified using discrete time models by simply doubling the time quantum. This is implemented by changing all transitions into two consecutive ones, that is, one transition in the new model takes half a unit, instead of one unit. This however, has two serious problems. One it adds a significant overhead to verification. The second one is that it is not possible to know by how much we should decrease the time quantum, because in general there is no way to find out when events that happened in different times have been considered simultaneous by the model. Because of this we cannot determine when the results of a verification using discrete time would be different if the model was refined. Our method does not suffer from these problems. There is no significant overhead added, since only one additional variable is created in the model, and all possible ordering of events are represented in the model, making refinements in the time quantum unnecessary.

6 Examples

6.1 Steam Boiler

In order to demonstrate the expressive power of our method we have verified the steam boiler example described in [14]. Steam boilers are mostly used in

thermoeletrical power plants. It is extremely important to keep a steam boiler working correctly since any malfunction may cause an accident with serious consequences. The system modeled consists of a water tank, two pumps, and sensors that measure the pumping rates, the steam evacuation rate and the water level. A controller oversees the operation of the system. The controller must guarantee that the water level is always between two values M_1 and M_2 at all times, and should try to maintain water level between the normal operating levels N_1 and N_2 as much as possible. The controller and the physical plant communicate in discrete intervals, once every Δ seconds. During each communication phase, all units send information to the controller, which responds by sending messages to the units. All communication takes place instantaneously.

The controller decides to turn the pumps on or off based on the water level w. The two pumps need five seconds to start pumping water in the tank because of the high pressure inside the tank. The pumps are turned off immediately after receiving a message to stop pumping from the controller. Four values are used by the controller to decide how many pumps should be active. Depending on these values and the current water level the controller turns one or both valves on or off (details about the system can found in [14]). We have modeled the high-level interactions between discrete control decisions and the continuous aspects of the underlying physical plant. We concentrate on the continuous aspects of the system and their modeling with the method described in the previous section.

We have set the values of the system constants as follows: sampling time $\Delta = 5$ seconds, maximal steam rate $W = 6$ liters per second, pumping capacity $P = 4$ liters per second, interval of normal water levels $[N_1 = 100, N_2 = 150]$ liters, interval of acceptable water levels $[M_1 = 25, M_2 = 200]$ liters. These constants have the same values as in [14], allowing direct comparison of results. In our model unit transitions model the passage of one second, and zero-length transitions are used to model nondeterministic events and decisions taken by the controller. Notice that verification can be performed very accurately, even though we use a coarse discretization of time.

The most important property of the steam boiler is that the water level is always between M_1 and M_2. We also require that the emergency-stop mode is never entered. Therefore, the unsafe states are those that satisfy the formula $(w < M_1) \vee (w > M_2) \vee emergency\_stop$. Using Verus, we have been able to verify that the controller maintains the water level within the required bounds. This result is the same obtained in [14]. The verification took 2.3 seconds and 1.1 MBytes of memory on a Pentium II system.

We have also verified other properties of the steam boiler using the *mincount* and *maxcount* algorithms. For example, an important parameter of the system is the size of Δ, the frequency of communication between controller and plant. Using Verus we have been able to determine that $\Delta = 6$ also satisfies the safety requirements, but $\Delta = 7$ does not. If communication between

controller and units is delayed by up to one second, safety is maintained, but longer delays can cause safety problems. Several other parameters have been identified, including, e.g., the minimum and maximum times needed for water to go from the minimum to the maximum level. The interval is $[20, \infty]$ seconds, meaning that the water may never reach the maximum water level from the minimum water level, but it never takes less then 20 seconds.

6.2 Automotive Engine Controller in Cutoff Model

In order to demonstrate the efficiency of the method we have verified an automotive engine controller in cutoff mode described in [17] and verified by HyTech. We have studied the cutoff mode, where we consider control of the engine once the driver has released the accelerator pedal. The system must then guarantee that the engine will deliver zero torque within a certain time. The control objective is to reach injection cutoff while minimizing acceleration discomfort. If fuel injection is abruptly cut off, the vehicle may exhibit very undesirable acceleration oscillations. If fuel injection remains on for a long time the car does not decelerate. In order to minimize these problems, the controller makes intelligent decisions about when and how to cut off fuel.

The system consists of the engine, which includes the driveline and the cylinders, and its controller. The engine has four cylinders, each of which cycles in lockstep through four phases in the following order: intake (I), compression (C), expansion (E), and exhaust. The controller must make its decision on injection (modeled by the binary output variable j) at the beginning of the preceding exhaust phase. If fuel is injected into a cylinder, the cylinder produces torque on its next expansion phase. Thus the driveline does not react to a control decision until three phases later.

The controller sets the value of j at each phase change, with the function F modeling the decision to inject fuel or not. The function F is defined over a transformed state space (over the variables x_1, x_2, x_3, x_4) that helps isolate the fundamental modes related to acceleration oscillations. Powertrain oscillations are due to the pair of complex conjugate poles, which are related to x_2 and x_3 components. Thus, our analysis concentrates on the $x_2 - x_3$ subspace, where the encirclements of the origin correspond to oscillations (more details about the system can be found in [17]).

The automotive engine controller should meet the requirement that for a given initial condition the state is close to the origin (injection cutoff) within a bounded number of phases (convergence). To show the convergence requirement using the same parameters described in [17] we have computed the maximum time from an initial state until a trajectory is close to the origin.

We have used Verus to verify the requirements. The code for the example has been generated automatically from the HyTech original code using a perl script written for this purpose. We have divided the $x_2 - x_3$ state space into 25 x 25 partitions increasing the accuracy of the rectangular approximations.

In our model, phase changes occur in unit time, and all other events happen in time zero. We have determined that the maximum time until a trajectory is close to the origin is 29 steps, the same result obtained by HyTech. Verification was performed very efficiently, but at the same time it has shown a limitation of our method. The source file for this example is extremely large, it has more than 250,000 lines of Verus code! It is, to the authors' knowledge, the largest example verified by symbolic model checking. It took Verus several hours to compile this code into a transition graph representing the system. Once the model was generated, however, verification was performed in only 18 seconds.

The reason for the long compilation time seems to be related to the fact that for systems which involve large constants, discretization can lead to a large state space representation even when using BDDs. This is caused by the binary encoding of integer values used. In some of these cases, continuous time models may be more efficient, since the representation is less dependent on time granularity. However, for models whose timing constants are well-behaved, a discrete-time model with a uniform BDD-based representation can present significant gains in efficiency. In this case it seems that both effects were present. The values represented for x_2 and x_3 are well behaved, but their values are large, as well as the number of operations that have to be performed on them, making the generation of the model slow, but possible. Verification, on the other hand, was performed extremely fast, showing that the complexity is related to the manipulation of large integer values, not to the representation of time.

7 Conclusions

In this work we propose a new algorithm to perform quantitative timing analysis of models that is more efficient than its predecessor. This algorithm, called *condition counting*, counts the minimum and maximum number of occurrences of events between two events *start* and *final*. The algorithm is used to implement an alternative method to represent time which enables the verification of systems that were previously considered to require dense time models. Verification under the new model can be performed as efficiently as for discrete time models. The proposed method has been implemented in Verus, but it can be used in most BDD-based symbolic model checkers. Two examples that had previously been verified by the dense-time tool HyTech have been modeled and verified in Verus. Future work includes a more accurate characterization of the expressive power of the method.

Acknowledgments

We would like to thank Howard Wong-Toi for the many useful discussions about the examples that have been verified in HyTech.

References

[1] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking for real-time systems. In *Proc. 5th Annual IEEE Symp. on Logic in Computer Science*, pages 414–425, Philadelphia, PA, USA, June 1990. IEEE Press.

[2] Eugene Asarin, Oded Maler, and Amir Pnueli. On discretization of delays in timed automata and digital circuits. In D. Sangiorgi and R. de Simone, editors, *CONCUR'98: Concurrency Theory.* 8^{th} *Int. Conf. Proc.*, volume 1466 of *LNCS*, pages 470–484, Nice, France, September 1998. Springer.

[3] S. Campos, E. Clarke, W. Marrero, and M. Minea. Verifying the performance of the PCI local bus using symbolic techniques. In *Proc. IEEE Int. Conf. on Comput. Design*, pages 72–78, Austin, TX, USA, October 1995. IEEE Press.

[4] S. V. Campos. *A Quantitative Approach to the Formal Verification of Real-Time Systems.* PhD thesis, School of Computer Science, Carnegie MellonUniv., 1996.

[5] S. V. Campos, E. M. Clarke, W. Marrero, and M. Minea. Timing analysis of industrial real-time systems. In *Proc. Workshop on Industrial-strength Formal Specification Techniques*, pages 97–107, Boca Raton, FL, April 1995. IEEE Press.

[6] S. V. Campos, E. M. Clarke, W. Marrero, M. Minea, and H. Hiraishi. Computing quantitative characteristics of finite-state real-time systems. In *Proc.* 15^{th} *IEEE Real-Time Systems Symp.*, pages 266–270, San Juan, Puerto Rico, December 1994. IEEE Press.

[7] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logic of Programs: Workshop*, volume 131 of *LNCS*, pages 52–71, Yorktown Heights, NY, USA, 1981. Springer.

[8] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[9] H. De-Leon and O. Grumberg. Modular abstractions for verifying real-time distributed systems. *Formal Methods in System Design*, 2:7–43, 1993.

[10] David L. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 197–212, Grenoble, France, June 1989. Springer.

[11] T. A. Henzinger, P. H. Ho, and H. Wong-Toi. HyTech: the next generation. In *Proc. 16th IEEE Real-Time Systems Symp.*, pages 56–65, Pisa, Italy, December 1995. IEEE Press.

[12] Thomas A. Henzinger, Zohar Manna, and Amir Pnueli. What good are digital clocks ? In W. Kuich, editor, *Automata, Languages and Programming.* 19^{th}

International Colloquium Proceedings, volume 623 of *LNCS*, pages 545–558, Wien, Austria, July 1992. Springer.

[13] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. In *Proc. 7th Annual IEEE Symp. on Logic in Computer Science*, pages 394–406, Santa Cruz, CA, USA, June 1992. IEEE Press.

[14] Thomas A. Henzinger and Howard Wong-Toi. Using HyTech to synthesize control parameters for a steam boiler. In *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, volume 1165 of *LNCS*, pages 265–282. Springer, 1996.

[15] K. G. Larsen, P. Pettersson, and W. Yi. Compositional and symbolic model-checking of real-time systems. In *Proc. 16th IEEE Real-Time Systems Symp.*, pages 76–87, Pisa, Italy, December 1995. IEEE Press.

[16] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[17] Tiziano Villa, Howard Wong-Toi, Andrea Balluchi, Joerg Preussig, Alberto Sangiovanni-Vincentelli, and Yosinori Watanabe. Formal verification of an automotive engine controller in cutoff mode. In *CDC98: IEEE Conference on Decision and Control*, Tampa, Florida, December 1998.

[18] S. Yovine. Kronos: A verification tool for real-time systems. *Springer International Journal of Software Tools for Technology Transfer*, 1, October 1997.